XMill: an Efficient Compressor for XML Data

Hartmut Liefke*
Univ. of Pennsylvania
liefke@seas.upenn.edu

Dan Suciu
AT&T Labs
suciu@research.att.com

Abstract

We describe a tool for compressing XML data, called `XMill`, that usually achieves about twice the compression ratio of `gzip` at roughly the same speed. The intended applications are XML data exchange and archiving. `XMill` does not need schema information (such as a DTD or an XML-Schema), but can exploit hints about such a schema in order to further improve the compression ratio. `XMill` incorporates and combines existing compressors in order to compress heterogeneous XML data: it uses `zlib`, the library function for `gzip`, as well as a collection of datatype specific compressors. `XMill` can be extended with new specialized compressors: this is useful in applications managing XML data with highly specialized data types, such DNA sequences, images, etc. The paper presents a theoretical justification for the method used, `XMill` architecture and implementation, a new languages for expression the hints about the XML schema, and a series of experiments validating `XMill` on several real data sets.

1 Introduction

We have implemented a compressor/decompressor for XML data, to be used in data exchange and archiving, called `XMill`. It achieves about twice the compression rate of general-purpose compressors (`gzip`), at about the same speed[1].

XML data is self-describing, a feature that inflates the data but confers flexibility. XML can describe a wide range of data kinds, from regular to irregular, from flat to deeply nested, from tree shaped to graph shaped. From a database perspective, XML is an instance of *semistructured data*, a data model proposed in [9] for self-describing, irregular data. Early research on semistructured data has focused on data models and query languages [1, 7] and on schemas [6, 18], and some of this research has been migrated to XML [13, 17].

While other data exchange formats preceded XML many years, XML is more likely to become universal because of its close relationship to the Web and because of the commitment of several major software vendors to integrate XML in their products. Its strength comes from its universal acceptance, not from particular technical merits. XML is now being adopted by many organizations and industry groups, e.g. in healthcare, banking, chemical, and telecommunications.

There are some serious concerns however about exporting one's data into XML. Since XML data is irregular and verbose, it can impact both query processing and data exchange. Many applications (e.g. Web logs, biological data, etc) use other, specialized data formats to archive and exchange data, which are much more economical than XML.

Given this fact, previous work on semistructured data has addressed the query processing problem. The solutions proposed come from various angles: designing a query processor from scratch [12, 15, 17], using relational databases [14, 16, 31], and using object-oriented databases [10, 24]. An important lesson has been learned from this work: it is essential to exploit any regularity in the XML data and leverage existing technologies whenever possible, to gain efficiency and scalability.

This paper addresses the XML exchange and archiving problems. We describe a compressor (`XMill`) and decompressor (`XDemill`)[2], to compress XML data for the purpose of exchange and archiving. We do not propose a new compressing algorithm, but rather design an architecture which leverages existing compressing algorithms

*This work was done while the author was visiting AT&T Labs.

[1] In an early stage of the project we called our compressor XMLZIP, but we had to change the name due to a copyright conflict.

[2] Both tools will be released for general use, pending AT&T internal approval.

and tools to compress XML data. Moreover, XMill is *extensible*, such that users can add their own specialized compressors if they have complex, application specific data types in their XML data. XMill is based on gzip (we use the function library version zlib), in addition to a few simple, data type specific compressors.

As with previous work in semistructured data, XMill is designed to exploit any regularity in the data in order to improve compression. Such regularity can be described manually by the user in the form of hints to XMill[3]. But they are not required and, when present, they do not constrain the input data in any way. This fits nicely the typical semistructured data scenario. An application can start using XMill immediately, with its default settings, to compress XML data: this is already a win over general-purpose compressors. As knowledge about the XML data is gained, for example because the two partners involved in the data exchange agreed on a common DTD or XML-Schema, XMill's settings can be tuned to achieve even better compression. If the data's structure evolves over time, some settings may become obsolete, slightly degrading the compression ratio (but still keeping it above that of general-purpose compressors). Importantly, no XML data will ever be rejected by XMill, no matter how inaccurate the hints.

Perhaps the most striking discovery we made with XMill is that by migrating data from other formats to XML the size of the compressed data decreases! Many such formats are in use today, for biological data, for scientific data, for Web logs, etc. In each case the data is stored as an ASCII file, in a simple (but application specific) format, usually designed to be reasonably space-efficient for the application at hand. When translated into XML the data usually expands (by a factor of 1.5 to 3 in our experiments), mainly because XML tags are verbose and must be repeated. The XML data typically compresses well with gzip, but is still larger than the original gzipped data (by a factor of 1.2 to 1.4). Until now, one had to pay a price in space for XML's added flexibility. With XMill however, the XML-ized data is compressed better than the original gzipped data (almost to half the size). This is because XML exposes the individual data items in an application-independent way, which makes it possible for a general-purpose tool like XMill to improve compression. Of course, the same kind of compression can be applied to the original format, but one has to write a specific compressor for each format. Thus, by converting to XML, one gains both flexibility and efficiency (when compression is used).
Our compressor XMill applies **three principles** to compress XML data:

Separate structure from data The *structure* consists of XML tags and attributes; it forms a tree. The *data* consists of a sequence of items (strings) representing element contents and attribute values. The structure and the data are compressed separately.

Group data items with related meaning Data items are grouped into *containers*, and each container is compressed separately. For example, all <name> data items form one container, while all <phone> items form a second container. This is a generalization of a well-known principle in relational databases: column-wise compression is better than row-wise compression (see e.g. [23]).

Apply different compressors to different containers Some data items are text, others are numbers, while others may be DNA sequences. XMill applies different specialized compressors (*semantic compressors*) to different containers.

An original component of XMill are the *container expressions*, a concise language used for grouping data items in containers, and for choosing the right combination of semantic compressors. Grouping is specified with some generalized regular path expressions. Each expression specifies either a single container, or a set of containers: exactly how many depends on what tags are present in the XML data. The choice of semantic compression is specified by combining atomic compressors into more complex ones. This is quite useful when the XML data has complex data types, such as comma separated integers. Users type the container expressions on the XMill command line, or store them in a separate command-line file.

The implementation of regular path expressions in XMill posed an interesting performance challenge. The compressor has to evaluate each regular expression on (the path of) each data item in the XML file. We first tried to use deterministic finite state machines (DFAs) to represent the path expressions: this slowed down XMill considerably. The second solution we tried was to use some caching based on *DataGuides* [18]. This worked well in most cases, but on deeply nested and irregular data (we had one instance of such data, Treebank [25]) the DataGuide grows unacceptably large, quickly exhausting the available main memory. We propose a novel solution based on on *reversed* DataGuide combined with pruning. The amount of necessary main memory used by the

[3]Such descriptions can also be extracted from an XML schema language (such as XML-Schema [33, 5]).

pruned, reversed DataGuide is small for most practical cases, and we also observed a significant compression time improvement over DataGuides.

We validated XMill on a variety of real data sets, comparing both its compression rate and speed to that of other compressors, We found that XMill consistently compressed better than gzip, on which it is based: by a factor of 2 for "data-like" XML, slightly less for "text-like" XML. It compressed far better than compress, a Unix tool which trades compressing rate for speed, and it even compressed better than bzip, a tool which achieves very good compression rates but is extremely slow. Both XMill's compression and decompression speeds were about the same as those of gzip and gunzip.

Applicability and limitations The compressor described here has two limitations. The first is that it is not designed to work in conjunction with a query processor. Previous work in database compression focused on the integration of the decompressor with the query engine, under the assumption that a smaller, compressed database can be processed faster. Our setting here is different, since our targeted applications are data exchange, where compression is used to better utilize network bandwidth, and data archiving, where compression is used to reduce space requirement. Some form of processing on the compressed XML data is possible, since XMill uses an 8 MByte window for compression, but more work is needed to determine how feasible that is. A second limitation of XMill is that it wins over existing techniques only if the data set is large, typically over 20KByte. Hence it is of limited or no use in XML messaging, where many small-sized XML messages are exchanged between applications.

Contributions In this paper, we make the following contributions.

- We describe an extensible architecture for an XML compressor which leverages existing compression techniques and semantic compressors to XML data. Users can extend it with application specific semantic compressors.
- We describe container expressions, a brief yet powerful language for grouping data items according to their semantics, and specifying combined semantic compressors.
- We present an efficient implementation technique for the path language, which dramatically improved performance for deeply nested data sources.
- We evaluate XMill on several real data sets and show that it achieves best overall compression rates among several popular compressors.
- We show that by using XMill one decreases the size of the compressed data by migrating from other data formats to XML.
- We give an information-theoretic justification for our approach to XML compression.

The paper is organized as follows. Sec. 2 describes two motivating examples. Sec. 3 provides background about compression techniques and gives an information-theoretic justification for our approach to XML compression. The architecture of XMill, the container expression language and semantic compressors are described in Sec. 4. In Sec. 5 we show several implementation techniques to make XMill scalable and achieve compression/decompression times that are competitive with gzip. Sec. 6 describes experimental results, which we discuss in Sec. 7. We describe related work in Sec. 8 and conclude in Sec. 9.

2 Motivating Example

We start by illustrating with a very simple, but quite useful example: Web Log files. Virtually every Web server logs its traffic, for security purposes, and this data can be (and often is) analyzed. Each line in the log file represents an HTTP request. A typical entry in such a log file is[4]:

```
202.239.238.16|GET / HTTP/1.0|text/html|200|1997/10/01-00:00:02|-|4478
|-|-|http://www02.so-net.or.jp/|Mozilla/3.01 [ja] (Win95; I)
```

[4]This is one line in the log file.

Different formats are currently in use: in our example we use a variation on Apache's *Custom Log Format*[5]. Each line is a record with eleven fields delimited by `|`: host, request line, content type, etc. Hence, the file's structure is very simple, with records with a fixed number of variable-length fields[6].

Collected over long periods of time, Web logs can take huge amounts of space. In our example we only considered a file with 100000 entries as the one above. Its size is almost 16MB, and it shrinks to 1.6MB after compressing with `gzip`:

```
weblog.dat:      15.9MB
weblog.dat.gz:    1.6MB
```

Applications processing such Web logs are brittle, and in general not portable, since different vendors use different formats. Even the same server can be configured to generate different log formats (for example to include more fields). To gain flexibility, we may consider converting the Web log into XML with the following format:

```
<apache:entry>
   <apache:host>202.239.238.16</apache:host>
   <apache:requestLine>GET / HTTP/1.0</apache:requestLine>
   <apache:contentType>text/html</apache:contentType>
   <apache:statusCode>200</apache:statusCode>
   <apache:date>1997/10/01-00:00:02</apache:date>
   <apache:byteCount>4478</apache:byteCount>
   <apache:referer>http://www02.so-net.or.jp/</apache:referer>
   <apache:userAgent>Mozilla/3.01 [ja] (Win95; I)</apache:userAgent>
</apache:entry>
```

Applications are now easy to write, since they can recognize both the field names and the server type. However the size increases substantially, both for the XML file and for its compressed version:

```
weblog.xml:      24.2MB
weblog.xml.gz:    2.1MB
```

Our goal is to gain from XML's flexibility without using more space. An obvious idea for saving space is to assign integer codes `(1, 2, 3, ...)` to the XML tags, and use a unique character for closing tags. A more interesting idea is to separate the XML tags (now encoded by numbers) from the data values, and compress with `gzip` independently the tags and the data values. We save space, because the XML tags are the same for each record, and `gzip` can encode this very efficiently (we review `gzip`'s algorithm in Sec. 3.2). With `XMill` this effect is accomplished by:

```
xmill -p // weblog.xml  weblog1.xmi
```

(We describe `XMill`'s command line in Sec. 4.2.) This brings the size down to:

```
weblog1.xmi:      1.75MB
```

which is better than `gzip`, but not as good as the original gzipped file.

The next idea is to compress data values separatedly, based on their tags: that is, all host values are compressed together, all request lines are compressed together, etc. We save more space, because `gzip` achieves better compression when applied to values of similar types, than when applied to a mixed stream of values. This behavior is the default in `XMill`, and can be achieved by:

```
xmill weblog.xml  weblog2.xmi
```

This reduces the size even further:

```
weblog2.xmi:      1.33MB
```

[5] `http://www.apache.org/docs/mod/mod_log_config.html`

[6] Missing values are common and are indicated by `-`.

```
-p//apache:host=>seqcomb(u8 "." u8 "." u8 "." u8)
-p//apache:userAgent=>seq(e "/" e)
-p//apache:byteCount=>u
-p//apache:statusCode=>e
-p//apache:contentType=>e
-p//apache:requestLine=>seq("GET " rep("/" e) " HTTP/1." e)
-p//apache:date=>seq(u "/" u8 "/" u8 "-" u8 ":" di ":" di)
-p//apache:referer=>or(seq("file:" t) seq("http://" or(seq(rep("." e) "/" rep("/" e)) rep("." e))) t)
```

Figure 1: XMill settings settings.pz for efficient compression of Web log data.

Note that we now use less space than the original gzipped file.

We can do quite a lot better than that. The idea is to inspect carefully each field and use a specialized compressor for it. For example the <apache:host> is usually (or always) an IP address, hence can be stored as four unsigned bytes; <apache:date> can also be stored more efficiently in binary. Most entries in <apache:requestLine> start with GET and end in HTTP/1.0 (some in HTTP/1.1): these substrings can be factored out. Other improvements are also possible. We decided to analyze eight of the eleven fields and applied specialized compressors available in XMill. The corresponding XMill command line is:

```
xmill -f settings.pz   weblog.xml  weblog3.xmi
```

where the file settings.pz is shown in Fig. 1 (specialized compressors are described in Sec. 4.3). With these settings we reduce the compressed size to:

```
weblog3.xmi:      0.82MB
```

Note that this is about half the original gzipped file. This achieves our goal: the compressed XML-ized data can be stored in less space than the compressed original data, while applications gain in flexibility[7].

The Web log example is quite simple, and illustrates column-wise compression applied to XML. The second example is much more complex, and is taken from biological databases. SwissProt is a well-maintained database for representing protein structure (http://www.expasy.ch/sprot/). It uses a specific data format, called EMBL [21], for representing information about genes and proteins. Fig. 2 shows an entry in the SwissProt database. Lines start with a two-letter code describing the meaning of the line: for example, AC is *accession number*, OC describes the *organism classifier*, etc. Some fields have additional structure, like CHAIN, which is encoded in the indentation.

We repeated the experiments above on a fragment of the SwissProt data[8]. Our file had 98MB, while gzip reduced it to 16MB:

```
swissprot.dat:     98MB
swissprot.dat.gz:  16MB
```

We chose a nested structure for the XML-ized SwissProt data, as shown in Fig. 3. This inflated the size to:

```
swissprot.xml:    165MB
swissprot.xml.gz:  19MB
```

Repeating the steps above we obtained the following improvements in size:

```
swissprot1.xmi    15MB
swissprot2.xmi    11MB
swissprot3.xmi     8.6MB
```

The first file is obtained by compressing the XML tags separatedly from the actual data. The second file is obtained by compressing the data values separatedly, according to their tags (the default in XMill). Finally, the last file is obtained after fine-tuning XMill on the SwissProt data.

In both examples the three steps correspond precisely to the compression principles spelled out in Sec. 1. As the examples suggests, each principle contributes with a significant improvement.

[7] Of course, an application has to decompress the data first.

[8] We omitted comments and the actual DNA sequence, which can be compressed using specialized compressors.

```
ID   108_LYCES      STANDARD;      PRT;   102 AA.
AC   Q43495;
DT   15-JUL-1999 (Rel. 38, Created)
DT   15-JUL-1999 (Rel. 38, Last sequence update)
DT   15-JUL-1999 (Rel. 38, Last annotation update)
DE   PROTEIN 108 PRECURSOR.
OS   Lycopersicon esculentum (Tomato).
OC   Eukaryota; Viridiplantae; Streptophyta; Embryophyta; Tracheophyta;
OC   euphyllophytes; Spermatophyta; Magnoliophyta; eudicotyledons;
OC   core eudicots; Asteridae; euasterids I; Solanales; Solanaceae;
OC   Solanum.
RN   [1]
RP   SEQUENCE FROM N.A.
RC   STRAIN=CV. VF36; TISSUE=ANTHER;
RX   MEDLINE; 94143497.
RA   CHEN R., SMITH A.G.;
RL   Plant Physiol. 101:1413-1413(1993).
DR   EMBL; Z14088; CAA78466.1; -.
DR   MENDEL; 8853; LYCes;1133;1.
KW   Signal.
FT   SIGNAL        1     30       POTENTIAL.
FT   CHAIN        31    102       PROTEIN 108.
FT   DISULFID     41     77       BY SIMILARITY.
FT   DISULFID     51     66       BY SIMILARITY.
FT   DISULFID     67     92       BY SIMILARITY.
FT   DISULFID     79     99       BY SIMILARITY.
SQ   SEQUENCE   102 AA;  10576 MW;  AFA4875A CRC32;
     MASVKSSSSS SSSSFISLLL LILLVIVLQS QVIECQPQQS CTASLTGLNV CAPFLVPGSP
     TASTECCNAV QSINHDCMCN TMRIAAQIPA QCNLPPLSCS AN
//
```

Figure 2: Example of SwissProt data.


```
<Entry id="108_LYCES" class="STANDARD" mtype="PRT" seqlen="102">
   <AC>Q43495</AC>
   <Mod date="15-JUL-1999" Rel="38" type="Created"></Mod>
   <Mod date="15-JUL-1999" Rel="38" type="Last sequence update"></Mod>
   <Mod date="15-JUL-1999" Rel="38" type="Last annotation update"></Mod>
   <Descr>PROTEIN 108 PRECURSOR</Descr>
   <Species>Lycopersicon esculentum (Tomato)</Species>
   <Org>Eukaryota</Org> <Org>Viridiplantae</Org>  ...  <Org>Solanum</Org>
   <Ref num="1" pos="SEQUENCE FROM N.A">
      <Comment>STRAIN=CV. VF36</Comment>
      <Comment>TISSUE=ANTHER</Comment>
      <DB>MEDLINE</DB>
      <MedlineID>94143497</MedlineID>
      <Author>CHEN R</Author> <Author>SMITH A.G</Author>
      <Cite>Plant Physiol. 101:1413-1413(1993)</Cite>
   </Ref>
   <EMBL prim_id="Z14088" sec_id="CAA78466"></EMBL>
   <MENDEL prim_id="8853" sec_id="LYCes" status="1133"></MENDEL>
   <Keyword>Signal</Keyword>
   <Features>
     <SIGNAL from="1" to="30">    <Descr>POTENTIAL</Descr>     </SIGNAL>
     <CHAIN from="31" to="102">   <Descr>PROTEIN 108</Descr>   </CHAIN>
     <DISULFID from="41" to="77"> <Descr>BY SIMILARITY</Descr> </DISULFID>
     ...
   </Features>
</Entry>
```


Figure 3: XML Representation of SwissProt entry

3 Background

3.1 XML

For the purpose of this paper, an XML document consists of three kinds of tokens: tags, attributes, and data values. Consider the following example:

```
<Book> <Title lang="English"> Data Compression </Title>
       <Year> 1995 </Year>
</Book>
```

Here Book, Title, Year are *tags*. Each tag occurs in pairs, a begin-tag (e.g. <Book>) and an end-tag (e.g. </Book>), and delimits an *element*. The text between the tags is called the element's content, and can consist of other elements and/or *data values*. In our example the Book's content are the elements Title and Year, while the Year's content is the data value 1995. The data values are always strings, but they sometimes are encodings of specific data types, like dates, integers, etc. An element may have a set of attribute-value pairs: in our example, lang is an attribute of Title, and its value is English. We prefix attributes with @, like in @lang, to distinguish them from tags, a convention borrowed from XPath [11].

An XML document may also contain processing instructions (PI), comments, CDATA values, and a document type declaration (DTD). Those special sequences are treated separately in XMill and stored in a single data container that is compressed with gzip.

XML Tree As usual we model an XML document as a tree: nodes are labeled with tags or attributes, and leaves are labeled with data values. The *path* to a data value is the sequence of tags (and, possible of attributes) from the root to the data value node.

3.2 Compressors

General Purpose Compressors Most practical dictionary compressors are derived from Ziv and Lempel's 1977 paper [34], and form the LZ family of compressors[9]. We review here the original, LZ77 compressor, since it impacts the architecture of XMill. While parsing the input string, the LZ77 compressor keeps the text seen previously. At each new character read, the compressor looks ahead, trying to find the longest common substring in the text seen so far. For example, consider the following sequence of characters:

```
A B C D E F B C D E G
            ^
```

When the second B is read, the compressor finds that the subsequence B C D E occurs in the text seen so far and replaces it with a back-pointer of the form (-offset,length):

```
A B C D E F (-5,4) G
```

Of special importance for XMill is LZ77's behavior on periodic sequences. Consider:

```
A B A B A B A B A B A B A B A B C
    ^
```

When the second A is read the compressor searches for a substring occurring in the text seen so far. The longest such substring has length 14, extending well into the unread text. The compressed output is:

```
A B (-2, 14) C
```

Hence periodic sequences are compressed very efficiently. LZ77 uses a *sliding window*, a fixed amount of memory for the text seen so far and the lookahead buffer.

The popular general-purpose compression tool gzip uses LZ77 in combination with other techniques. A function library, zlib, makes the gzip functionality available to applications. We used zlib in XMill, and will refer to zlib and gzip interchangeably in the paper.

[9]The reversal of initials is a historical mistake, according to [4].

Special Purpose Compressors A variety of special-purpose compressors exists, ranging from ad-hoc to highly complex ones [4, 29]. Special data types can be encoded in binary, e.g. integer or dates. A *dictionary encoding* assigns an integer code to each new word in the input, and stores the mapping from codes to strings in a separate dictionary. *Differential encoding*, or delta encoding, is useful for numeric data with small variation. For example: `10200 10240 10185 10182 ...` will be encoded as: `10200 +40 -55 -3 ...` Complex compressors exists for a variety of specialized data types, e.g. images, sound, DNA sequences [4, 20, 2].

3.3 Information Theory

Claude Elwood Shannon developed in the late 40s and early 50s almost single handedly the theory of information. In his classic paper [32] he considers an *information source*, a *channel*, and a *destination*, and studies how much information can be sent by the source to the destination through the channel. In the case of a noiseless channel, this amount is given by how well one can compress the source. A source S generates a message $x_1, x_2, \ldots, x_m$, symbol by symbol, with each symbol drawn from a fixed, finite alphabet $A = \{a_1, \ldots, a_n\}$ associated to that source. For example, a binary source will have $A = \{0, 1\}$, an ASCII source may have $A = \{0, 1, \ldots, 127\}$, etc. Shannon modeled a source as a Markov Process, and defined the *entropy* of a source, as a measure of the uncertainty associated with that source. In the case of very simple Markov Processes (called order-0 models), in which each symbol a_i has a fixed probability p_i, Shannon's definition of the entropy is:

$$H \stackrel{\text{def}}{=} p_1 \log \frac{1}{p_1} + \ldots + p_n \log \frac{1}{p_n}$$

In the general case, a Markov Process has a fixed set of states, and each state has a different set of probabilities for the symbols $a_1, \ldots, a_n$, and for the transitions into the next state. (Formal definition omitted.) Shannon proved in his paper the *fundamental theorem for a noiseless channel*, which essentially says that, on average, one cannot encode a sequence with less than H bits per symbol, and that (almost) optimal encodings exists. Reformulated in the context of compression, the theorem says the following. A message of m symbols from a source with entropy H cannot be compressed to less than mH bits, on average. Conversely, there exists an almost optimal compressor, which compresses a message of length m to (almost) mH bits on average. As a simple illustration, an ASCII source with uniform probabilities ($p_1 = \ldots = p_{128} = 1/128$) needs $H = \log(1/128) = 7$ bits per character: when the distribution is skewed, less bits can be used. Dictionary compressors, discussed at the beginning of this section, have been shown to achieve almost optimal compression [4].

Optimal compression of XML and heterogeneous sources Unlike Shannon's information sources, XML data is heterogeneous. We give here a formal definition of a heterogeneous information source and prove that the three compression principles in Sec. 1 achieve optimal compression for heterogeneous sources.

A *heterogeneous information source* S is a collection of $k+1$ sources $S_0, S_1, \ldots, S_k$, over alphabets $A, B_1, \ldots, B_k$. The first alphabet has k symbols, $A = \{a_1, \ldots, a_k\}$, called *tags*, while the others can have an arbitrary number of symbols. The heterogeneous sources emits messages of the following shape:

$$x_1, y_1, x_2, y_2, \ldots, x_m, y_m \tag{1}$$

where $x_1, \ldots, x_m \in A$, and, whenever $x_j = a_i$, then the next symbol y_j belongs to B_i.

If all $k+1$ sources are of order 0, then the heterogeneous source S is equivalent to a (nonheterogeneous) source modeled by a particular Markov Process with $k+1$ states over the alphabet $A \cup B_1 \cup \ldots \cup B_k$ (details omitted).

Heterogeneous sources are a simplification of XML since they don't model nesting: nesting can be modeled by probabilistic grammars [4], but this is beyond the scope of this paper.

We show next that our three compression principles achieve optimal compression for messages like (1). To be precise, the compression proceeds in three steps: (1) separate the tags $x_1, x_2, \ldots$ from the data items $y_1, y_2, \ldots$, (2) further separate the data items according to their source $S_i, i = 1, k$, (3) apply an optimal compressor for each source $S_0, \ldots, S_k$. Let $H_0, H_1, \ldots, H_k$ be the entropies of the $k+1$ sources, and let $p_1, \ldots, p_k$ be the probabilities of source S_0. Then the compression just described uses:

$$mH_0 + mp_1H_1 + mp_2H_2 + \ldots mp_kH_k \tag{2}$$

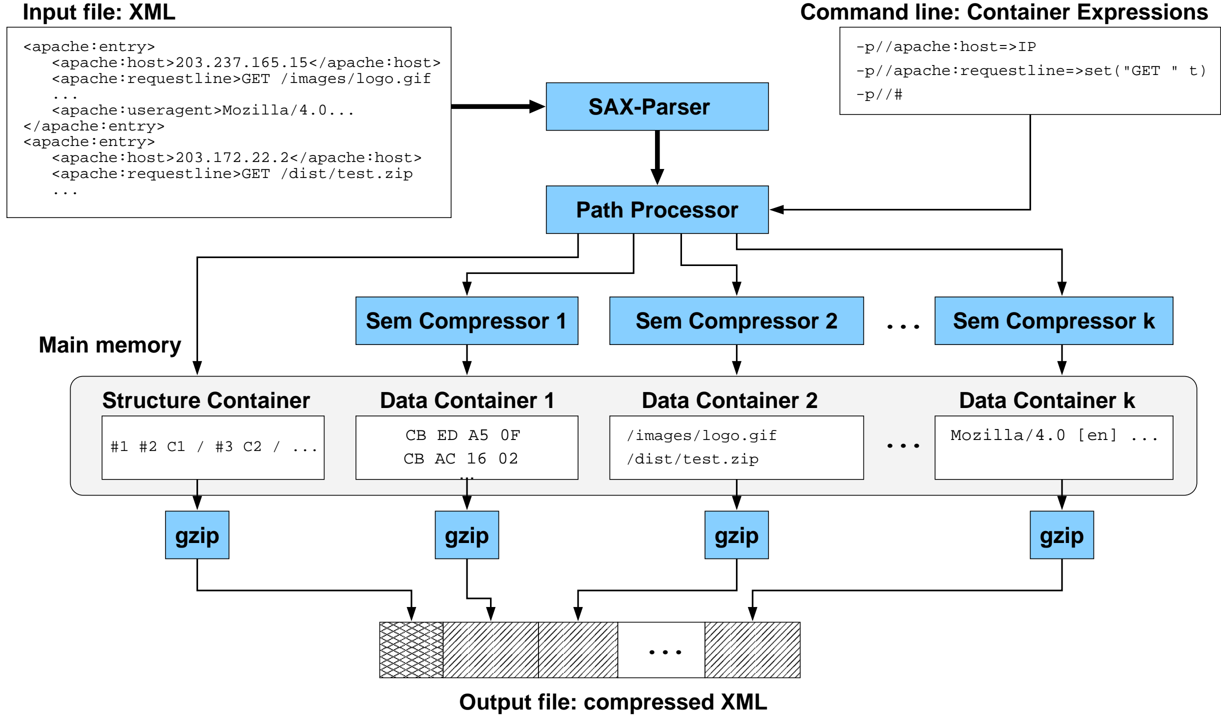


Figure 4: Architecture of the Compressor

bits for the message (1) of length $2m$. This is because it needs mH_0 bits for $x_1, x_2, \ldots, x_m$; then there are, on average mp_1 characters from source S_1, etc. Our theorem below proves that this is optimal:

Theorem 3.1 *The entropy of the heterogeneous source S is: $\frac{1}{2}(H_0 + p_1 H_1 + \ldots + p_k H_k)$. Hence the number of bits used in (2) is optimal on average.*

4 The Architecture of `XMill`

The architecture of `XMill` is based on the three principles described in Section 1 and is shown in Figure 4. The XML file is parsed by a SAX[10] parser that sends tokens to the path processor. Every XML token (tag, attribute, or data value) is assigned to a container. Tags and attributes, forming the XML structure, are sent to the structure container. Data values are sent to various data containers, according to the container expressions, and containers are compressed independently. Before entering the container, a data value may be compressed with an additional semantic compressor.

The core of `XMill` is the *path processor* that determines how to map data values to containers. The user can control this mapping by providing a series of *container expressions* on the command line. For each XML data value the path processor checks its *path* against each container expression, and determines either that the value has to be stored in an existing container, or creates a new container to hold that value. Finally, each container is compressed using `gzip` and then stored in the output file.

Containers are kept in a main memory window of fixed size (the default is 8MB). When the window is filled, all containers are gzipped, stored on disk, and the compression resumes. In effect this splits the input file into blocks that are compressed independently.

Users can associate semantic compressors with containers. A few *atomic* semantic compressors are predefined in `XMill`, like binary encoding of integers, differential compressors, etc. In addition, users can *combine* simple semantic compressors into more complex ones: this is useful when data values have lexical structure, e.g. integers

[10] SAX stands for *Simple API for XML*, `http://www.megginson.com/SAX/`.

separated by commas. Alternatively, users can write new semantic compressors and *link* them into `XMill`. This is useful when the XML data contains highly specialized types, like DNA sequences, for which special purpose compressors exists [20, 2].

Users, of course, do not have to specify any semantic compressor: the default *text* semantic "compressor" simply copies its input to the container, without any semantic compression.

The decompressor `XDemill` is simpler, and its architecture is not shown. After loading and unzipping the containers, the decompressor parses the structure container, invokes the corresponding semantic decompressor for the data items and generates the output. We explain next the compressor architecture in detail.

4.1 Separating Structure from Content

The *structure* of an XML file consists of its tags and attributes, and is tokenized in `XMill` as follows. Start-tags are dictionary-encoded, i.e. assigned an integer value, while all end-tags are replaced by the token `/`. Data values are replaced with their container number. To illustrate, consider the following small XML file:

```
<Book> <Title lang="English"> Transaction Processing </Title>
       <Author> Gray </Author>
       <Author> Reiter </Author>
</Book>
```

Its structure is best visualized by erasing each data value (we replace it with `C` followed by the container number):

```
<Book> <Title lang="C3"> C4 </Title> <Author> C5 </Author> <Author> C5 </Author> </Book>
```

Here we assumed that the `@lang` values are stored in the data container 3, the `titles` in container 4, and the `authors` in container 5. The dictionary encoding of tags and attributes and the resulting tokenized structure are:

```
Book = #1,   Title = #2,   @lang = #3,   Author = #4
Structure = #1 #2 #3 C1 / C2 / #4 C3 / #4 C3 / /
```

For readability, throughout the paper we represent tokens as strings. In practice all tokens are encoded as integers (with 1, 2, or 4 bytes, see Sec. 4.3): tags/attributes are positive integers, `/` is 0, and container numbers are negative integers. The structure above needs 14 bytes.

So far we have ignored white spaces between tags, e.g. between `<Book>` and `<Title>`, and the decompressor produces a standard indentation: this is sufficient for most applications. Optionally, `XMill` can preserve the white spaces faithfully: in that case it stores them in container[11] 1. In our example the structure becomes:

```
<Book> C1 <Title lang="C2"> C3 </Title> C1 <Author> C4 </Author> C1 <Author> C4 </Author> C1 </Book>
```

The size of the compressed file typically increases only slightly when white spaces are preserved: around 4%. For Treebank, a linguistic database (see Sec. 5), the increase is higher (30%) because of the deeply nested structure. In the rest of the paper we will assume that white spaces are ignored, unless otherwise specified.

Our encoding of the structure is very simple. Of course, one could imagine many more sophisticated encoding methods, e.g. by exploiting the document's DTD. We observed however that, in practice, our simple encoding scheme compresses extremely well using `gzip`, making more sophisticated techniques unnecessary. For example, assume a large collection of books, as the one above, and assume for simplicity that all share the same structure: a `Title`, a `@lang` attribute, two `Authors`. Then the structure container will consist of a large number of repeated sequences like:

```
#1 #2 #3 C1 / C2 / #4 C3 / #4 C3 / /
#1 #2 #3 C1 / C2 / #4 C3 / #4 C3 / /
. . .
```

[11] Container 0 holds the structure while container 2 holds the PI's, DTD's, and comments.

Recall from Sec. 3.2 that `gzip` compresses such strings extremely well. For example, assuming a large enough window, LZ77 compresses the structure of 10000 books into 16 bytes[12]. Good compression is obtained even when the XML structure has small variations, e.g. `books` with one, two, or three `authors`, missing `@lang`, etc. In our experiments the compressed structure was typically around 1%-3% of the compressed file, depending on the data source and semantic compressors. For data with highly irregular structure, the structure container can take more space (20% for Treebank).

4.2 Grouping Data Values Based on Semantics

Each data value is uniquely assigned to one data container. The mapping from data values to containers is determined by the following information: (1) the data value's path, and (2) the user-specified container expressions. We describe them next, using the following running example:

```
<Doc> <Book> <Title language="English"> Compression </Title>
             <Year> 1995 </Year>
      </Book>
      <Person> <Name>  Tom </Name>
               <Title> Mr. </Title>
               <Child> Tim </Child>
               <Child> Karen </Child>
      </Person>
</Doc>
```

Recall that the *path* to a data value is the sequence of tags from the root to that value (Sec. 3.1): e.g. the path to `Compression` is `/Doc/Book/Title`, while the path to `"English"` is `/Doc/Book/Title/@language`.

Container Expressions A natural idea is to create one container for each tag or attribute. For example all `Title` data values go to one container, all `@language` attribute values go to a different container, etc. Equivalently, the container is determined by the last tag (or attribute) in the path: using XPath regular expressions (see below) we have one container for `//Title`, one for `//@language`, etc.

This simple mapping from tags to containers performs well in most cases in practice, but sometimes it is too restrictive. The context may change the tag's semantics: `/Doc/Book/Title` has a different meaning from `/Doc/Person/Title`, hence the two `Title`'s are best compressed separately. Conversely, different tags may have the same meaning, like `Name` and `Child`.

Our approach is to describe mappings from paths to containers with container expressions. Consider the following regular expressions derived from XPath [11]:

```
e ::= label | * | # | e1/e2 | e1//e1 | (e1|e2) | (e)+
```

Except for `(e)+` and `#`, all are XPath constructs: `label` is either a `tag` or an `@attribute`, `*` denotes any tag or attribute, `e1/e2` is concatenation, `e1//e2` is concatenation with any path in between, and `(e1|e2)` is alternation. To these constructs we added `(e)+`, the strict Kleene closure.

The interesting novel construct is `#`. It stands for any tag or attribute (much like `*`), but each match of `#` will determine a new container. We defer the formal semantics of container expression to the end of this section. A *container expression* has the form `c ::= /e | //e`, where `/e` matches `e` starting from the root of the XML tree while `//e` matches `e` at arbitrary depth in the tree. We abbreviate `//*` with `//`.

Example 4.1 `//Name` creates one container for all data values whose path ends in `Name`. `//Person/Title` creates a container for all `Peron`'s titles. `//` places all data items into a single container.

Example 4.2 `//#` creates a family of containers: one for each ending tag or attribute. It is a concise way to express a whole collection of container expressions: `//Title`, `//@language`, `//Name`, etc. (one for each tag in the XML file).

[12] However `gzip` requires 331 bytes.

Example 4.3 `//Person/#` creates a distinct container for each tag under `Person`, `//#/#` creates a distinct container for the last two tags, and `(#)+` creates a distinct container for every path.

Container expressions `c1, ..., cn` are given in the command line, with the `p` switch:

```
xmill -p c1  -p c2  . . .  -p cn  file.xml  file.xmi
```

For each data value, the path processor matches its path against $c_1, c_2, \ldots$, in that order. Assuming the first match is found at c_i, the processor computes the "values" of the `#`'s in c_i which made the match possible. (Formal semantics below.) These values uniquely determine the data value's container.

The container expressions only need to be specified at the compressor's side.

Example 4.4 Consider two examples of command lines:

```
xmill -p //#   file.xml file.xmi
xmill -p //Person/Title -p //Person/(Name|Child) -p //# file.xml file.xmi
```

The first groups and compresses data values according to their ending tag. The second compresses all `Person`'s titles together, all `Person`'s names and children together, and all other data values are compressed based on their ending tag. In particular `/Doc/Book/Title` and `/Doc/Conference/Paper/Title` will be compressed together, and separately from `/Doc/Person/Title`. Note that the order matters.

Default Behavior The path expression `-p //#` is always inserted at the end of the command line. In particular, the command line:

```
xmill file.xml file.xmi
```

is equivalent to:

```
xmill -p //#  file.xml  file.xmi
```

This ensures that every data value is stored in at least one container, and provides a reasonable default behavior when the user does not specify any container expressions.

Formal Semantics Given a container expression c and a path p, the function $Match(c, p)$ in Fig. 5 computes a set of strings, denoting the possible assignments to `#`'s which result in a match. There is a match if and only if $Match(c, p) \neq \emptyset$. On ambiguous expressions [28] $Match(c, p)$ may return more than one match. In the figure ε denotes the empty string, and / denotes string concatenation, both of strings and of sets of strings. Formally, if c_i, $i = 1, n$, is the first container expression matching p, then the data value with path p will be stored in a container uniquely identified by $(i, Match(c, p))$.

$Match(/e, p) = Match(e, p)$	$Match(\#, l) = \{l\}$
$Match(//e, p) = \bigcup_{q/p_1=p} Match(e, p_1)$	$Match(e_1/e_2, p) = \bigcup_{p_1/p_2=p} Match(e_1, p_1)/Match(e_2, p_2)$
$Match(l, l) = \{\varepsilon\}$	$Match(e_1//e_2, p) = \bigcup_{p_1/q/p_2=p} Match(e_1, p_1)/Match(e_2, p_2)$
$Match(l_1, l_2) = \emptyset$ if $l_1 \neq l_2$	$Match(e_1 \mid e_2, p) = Match(e_1, p) \cup Match(e_2, p)$
$Match(*, l) = \{\varepsilon\}$	$Match((e)+, p) = \bigcup_{p_1/p_2/\ldots/p_n=p} Match(e, p_1)/\ldots/Match(e, p_n)$

Figure 5: Formal Definition of the Function $Match(c, p)$

Example 4.5 Consider the path $p =$ `/Doc/Conf/Paper/Title`. Then:

$$Match(/\texttt{Doc}/\texttt{Title}, p) = \{\} \qquad Match(//\#/\#, p) = \{\texttt{Paper/Title}\}$$
$$Match(/\texttt{Doc}//\texttt{Title}, p) = \{\varepsilon\} \qquad Match(/(\#)+, p) = \{\texttt{Doc/Conf/Paper/Title}\}$$
$$Match(//\#, p) = \{\texttt{Title}\} \qquad Match(//\#//\texttt{Title}, p) = \{\texttt{Doc,Conf,Paper}\}$$

Compressor	Description	Compressor	Description
t	default text compressor	**u**	compressor for positive integers
i	compressor for integers	**u8**	compressor for positive integers < 256
di	delta compressor for integers	**rl**	run-length encoder
e	enumeration (dictionary) encoder	**”...”**	constant compressor

Table 1: Atomic Semantic Compressors

4.3 Semantic Compressors

Our early `XMill` implemention did not have semantic compressors. We quickly realized however that certain values are not compressed very well by `gzip`. For example `gzip`'s compression of IP addresses didn't even come close to the standard 4 bytes per address. We soon discovered that XML data often comes with a whole variety of specialized data types like integers, dates, US states, airport codes, which are best compressed by specialized semantic compressors.

We distinguish three kinds of semantic compressors: atomic, combined, and user-defined.

Atomic semantic compressors: There are eight such compressors in `XMill`, shown in Table 1. We explain them next. For a more general description of semantic compressors, we refer to [29] or standard textbooks [30]. The text compressor `t` does not compress, but rather copies the string to the container unchanged (it will be compressed later by `gzip`). Positive integers (compressor `u`) are binary encoded as follows: numbers less than 128 use one byte, those less than 16384 use two bytes, otherwise they use four bytes. The most significant one or two bits describe the length of the sequence. Signed integers (compressor i) are stored similarly. The integer compressor `u8` stores a number between 0 and 255 in one single byte. The delta compressor for integers stores the difference between successive numbers and the run-length encoder encodes sequences of identical data values as (val,count)-pairs. The enumeration encoder `e` will assign a positive integer to each new data value and keep a dictionary of all data values seen before. Finally, the constant compressor does not produce any output (the best compression of all !), but checks that the input is the given constant. It is especially useful in combined compressors described below. Some semantic compressor-decompressor pairs may be lossy, e.g. `u, u8, i` do not preserve leading zeros.

Semantic compressors are specified on the command line as follows. The syntax of container expressions is extended to optionally include semantic compressors:

```
C ::= c | c=>s
```

where `c` is a container expression (Sec. 4.2) and `s` is a semantic compressor. When missing, the default semantic compressor is text. For a simple illustration, consider the example:

```
xmill -p //price=>i  -p //state=>e   -p //description//  file.xml file.xmi
```

The `price` data items are compressed as integers, `states` as enumeration values, and the text under `description` (at any depth) is placed in a single container with no semantic compression. All remaining data items are grouped based on their tag (recall that the default `-p //#` is added at the end), with no semantic compression.

A semantic compressor may reject its input string. In the example above, a `price` value which does not parse as an integer will be rejected by the `i` compressor. In that case `XMill` tries the next path expression: eventually, the last `-p //#` will cause the data value to be stored in some container. Thus, price values like `3999, 1450, low, 55, high, 1099, ...` will be stored in two distinct containers. The user can exploit this behavior by specifying different alternative semantic compressors, like in

```
xmill -p //price=>i  -p //price=>e  file.xml  file.xmi
```

where all non-numeric `prices` are dictionary encoded.

Combined compressors: Often data values have structure. For example an IP address consists of four integers separated by dots (e.g. `104.44.29.21`); a `request` value (Sec. 2) consists of `GET` followed by a variable string. `XMill` has three compressor combinators for compressing such values:

- *Sequence Compressor* `seq(s1 s2 ...)`. For example `seq(u8 "." u8 "." u8 "." u8)` compresses an IP address as four integers. To simplify parsing, the sequence compressor requires every other semantic compressor (i.e. `s2, s4, ...` or `s1, s3, ...`) to be a constant. A variation is `seqcomb(s1 s2 ...)`, described below.
- *Alternate Compressor* `or(s1 s2 ...)`. For example, consider page references in a biblography file. These can be either like `145-199`, or single pages like `145`. The composite compressor is `or(seq(u "-" u) u)`. A variation is `orcomb(s1 s2 ...)` described below.
- *Repetition Compressor* `rep(d s)`. Here `d` is the delimiter and `s` another semantic compressor. For example, a sequence of comma separated keywords can be compressed by `rep("," e)`.

Fig. 1 illustrates the use of combined semantic compressors for the Weblog data.

User-defined Compressors Some applications require highly specialized compressors, like for DNA sequences [20, 2]. Users can write their own compressors/decompressors and link them into `XMill` and `XDemill`, conforming to a specified API, called SCAPI (Semantic Compressor API, Sec. 5.3). The API requires the user to implement several C++ methods to identify the compressor with a unique name and to parse, compress, decompress values. The name can then be used in place of any atomic semantic compressor, as in:

```
xmill -p //DNAsequence=>dna  file.xml
```

where `dna` is the compressor's name. The extended `XMill` becomes application specific, since a file compressed with such an extended `XMill` can only be decompressed by an `XDemill` with the corresponding decompressor.

Sub-containers In the simplest case, each data value is stored in one container. Combined, or user-defined compressors may split an XML value string into several atomic values, and these can be stored either together or separately. To allow the user to have control over how subvalues are grouped, each container is split into sub-containers, and each combinator defines precisely how sub-containers are combined. Every atomic compressor in Table 1 defines one sub-container, except for the constant compressor which defines zero sub-containers. `seq(s1 s2 ...)` takes the disjoint union of all the sub-containers, e.g. `seq(u8, ".", u8, ".", u8, ".", u8)` has four sub-containers, while `seqcomb(s1 s2 ...)` overlaps the sub-containers, e.g. `seqcomb(u8, ".", u8, ".", u8, ".", u8)` has one sub-container. Similarly `or(s1 s2 ...)` takes the disjoint union, while `orcomb(s1 s2 ...)` overlaps them (both define an additional sub-container with the choice). The repetition combinator always overlaps the sub-containers.

5 Implementation

`XMill` and `XDemill` are implemented in C++, and have together about 12,000 of code. We describe here the main modules, refering to Fig. 4.

5.1 Parser

We wrote our own SAX parser for XML. SAX is a style of parsing using callbacks that translate the XML file into a stream of events: one event for each start-tag, one for each end-tag, and one for each data value; in addition, our parser also has events for attributes and attribute values. Every XML event (token) is sent to the path processor. This style of parsing has the advantage that it does not need to store a complete internal representation of the XML file, and makes it possible for `XMill` to interrupt the parsing anywhere when the memory window is full, and later resume it at the next token.

The parser has a fixed buffer of 64KB to store the current token, and data values which are longer are simply split into multiple tokens. To illustrate, recall the example in Sec. 4.1, where an `author` was described as `<Author> C5 </Author>` in the structure: an extremely long `author` name may be split into three values, and described as `<Author> C5 C5 C5 </Author>`. The semantic compressor is invoked for each value separately.

5.2 Path Processor

The path processor keeps track of the current path for each data value and evaluates successively each container expression on the path: the latter involves evaluating a regular expression, and, if successful, evaluating the semantic compressor on that data value. This is the most time-critical piece in the compressor and we tried three different evaluation methods.

Direct Evaluation of Regular Expressions Each container expression is translated into a minimized, deterministic automaton (DFA)[22]. For an illustration, Fig. 6 (a) shows two regular expressions and their corresponding DFAs. Note that the wildcards `#` and `*` are considered to be symbols (more on that below). The automata are ordered according to order in the command line. While parsing the XML file, the current state of each DFA is maintained and changed whenever a start- or end-tag is parsed. For each received data value, the first container expression with a final current DFA state is matched. Even though the representation as DFAs allows an efficient evaluation, it is rather inefficient for evaluating many container expressions.

Evaluation using DataGuides DataGuides were introduce in [18] as a concise and accurate summary of a semistructured data instance. We review here their definition, in the simpler context of XML data. Let $p_1, p_2, \ldots$ be all paths occurring in an XML file (this set is closed under prefixes). The *DataGuide* for that file is the trie structure for $p_1, p_2, \ldots$ More precisely, the DataGuide is a tree, with one node for each distinct path, and an edge labeled l between any two node of the form p and $p.l$, where p is a path and l is a label. For example, the paths in the Web log data are:

```
/
/apache:entry
/apache:entry/apache:host
/apache:entry/apache:requestLine
. . .
```

and the DataGuide is represented in Fig. 6(b). `XMill` also maintains at each DataGuide node an ordered list of DFA states (represented with smaller circles in Fig. 6(b)). The DataGuide and its associated lists are constructed incrementally, as the XML data is processed. The DataGuide acts like a cache, memorizing the set of automata states for each path: it speeds up processing, and at the same time could be discarded at any time (e.g. when the memory is exhausted), either entirely, or partially.

We found this evaluation strategy to be very efficient on all data sets except for the most irregular and deeply nested data. For example, the DataGuide for the Weblog data had 11 nodes, while that for the SwissProt data had 35 nodes.

Evaluation using Reversed DataGuides Irregular and deeply nested data causes the DataGuide to grow out of proportions. An example of such data is the XML-ized *TreeBank* linguistic database[13] [25], which contains annotated sentences from the Wall Street Journal. Fig. 7 shows a fragment of the XML data corresponding to the sentence *"A stockbroker is an example of a profession in trade and finance."*. Each XML-tag has a specific linguistic meaning, e.g., `<NP>` denotes *noun-phrases*, `<CC>` represents *coordinating conjunction*, and so on. Figure 8(a) shows the DataGuide for the sentence. The DataGuide had 340000 nodes, which translated into about 16MB of main memory[14], far exceeding our 8MB memory window. A possible solution is to flush the DataGuide when it increases too much (this would slow down processing). Another is to collapse equivalent nodes in the DataGuide, i.e. those which have identical lists of states (this would be costly to compute).

We propose a new cache strategy based on *reversed DataGuides*, which is just the trie structure for the reverse paths. Fig. 8(b) shows the reversed DataGuide for the Treebank fragment in Fig. 7. To make reversed DataGuides work, we also compute the reversed DFA's for the regular expressions, which enable us to parse paths in reverse: the reversed DataGuide's nodes is then annotated with lists of states in the reversed DFA's, as before.

Reversing in itself is not the solution to our problem: the reversed DataGuide in our example has 1.1 million nodes (in contrast to 340000 nodes). What makes it work is that it is possible to *prune* the reversed DataGuide.

[13] More information about TreeBank is available under `http://www.cis.upenn.edu/~treebank/`.

[14] Recall that each node is annotated with references to the DFA states.

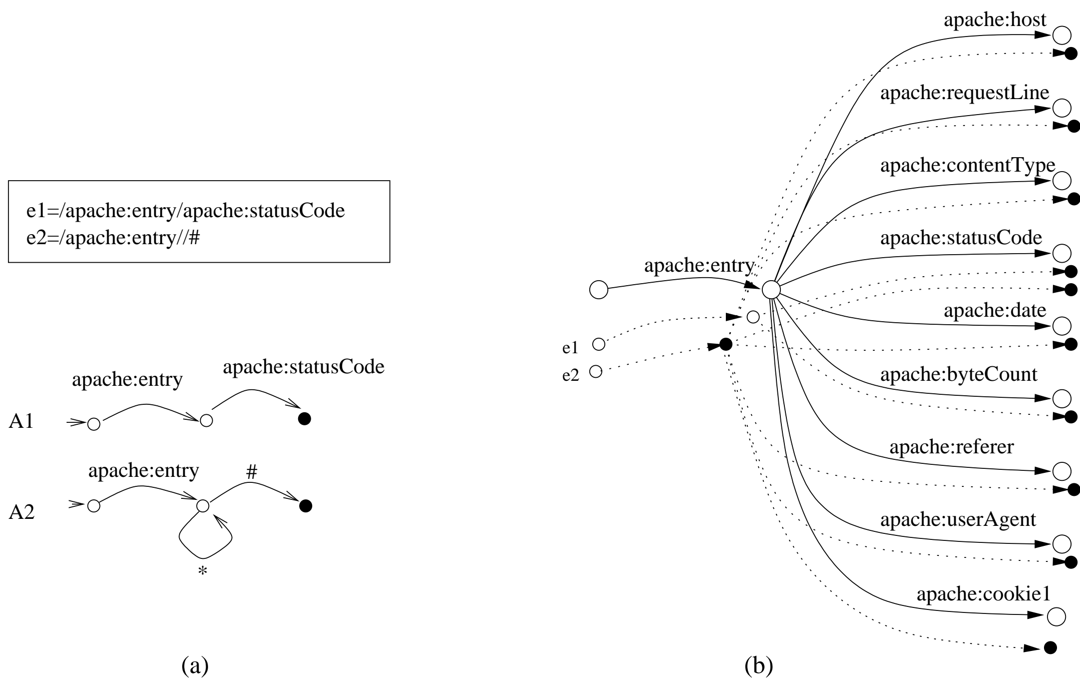


Figure 6: The DataGuide for the Weblog data

```
<S> <NP>                                    .
     <DT>A</DT>                             .
     <NN>stockbroker</NN>                 <PP>
    </NP>                                  <IN>in</IN>
    <VP>                                   <NP>
     <VBZ>is</VBZ>                          <NN>trade</NN>
     <NP>                                   <CC>and</CC>
      <DT>an</DT>                           <NN>finance</NN>
      <NN>example</NN>                     </NP>
      <PP>                                </PP>
       <IN>of</IN>                       </NP>
       <NP>                             </PP>
        <DT>a</DT>                     </NP>
        <NN>profession</NN>           </VP>
        .                             <.>.</.>
        .                            <S>
```

Figure 7: A Sentence in the Treebank Database

The observation here is that container expressions usually discriminated based on the last few tags in the path: for example `//NP/CC` and `//#` only look at the last one or two tags. The continues edges in Fig. 8 (b) show the pruned reversed DataGuide for these two regular expressions: they form only a small fraction.

How much can be pruned depends on the regular expressions. In all our examples the reversed DataGuide were pruned after one or two tags. For the Treebank data pruning was done after one tag, reducing the reversed DataGuide to approx. 100 nodes (the number of distinct tags in the XML file). This is a dramatic memory saving.

We ran `XMill` on the Treebank data with 3 container expressions. The direct implementation took 32s; the implementation with DataGuides (340,000 nodes) took 30s (and 16MB memory for the DataGuide); the implementation with reversed DataGuides took 26s. For 12 container expressions, the direct implementation

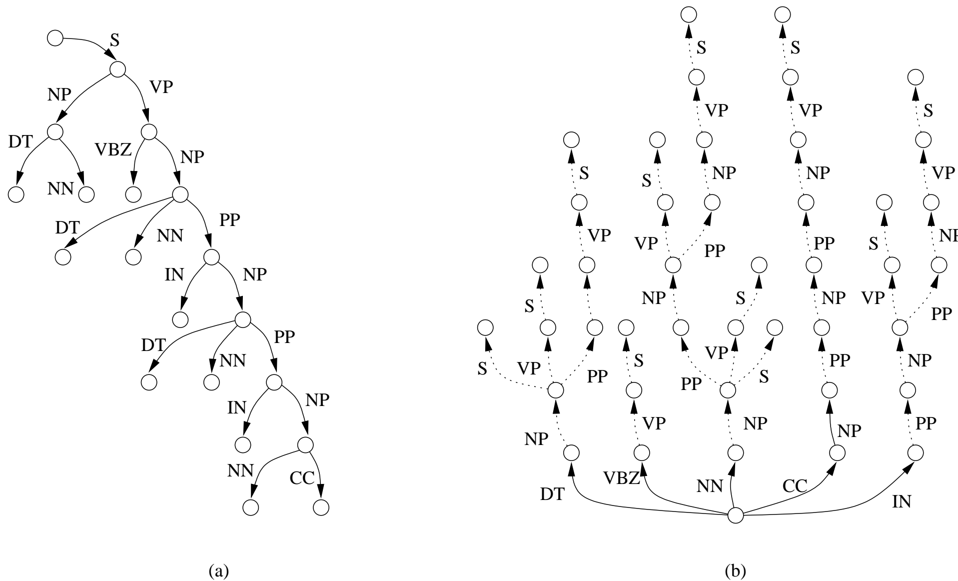


Figure 8: The Forward (a) and Reversed (b) DataGuide for a Fragment of TreeBank

becomes slower (41s) and the DataGuide implementation will require more memory (29MB). Only the reversed DataGuide implementation will run at about same speed with a very small DataGuide.

Dealing with #: approximating $Match(c, p)$ Given a container expression c we need to compute the function $Match(c, p)$ in order to determine the exact container for the path p (Sec. 4.2). A computation according to its definition (Fig.5) is expensive, and we settled for an approximation which works well in practical cases. First, when we translate regular expressions into DFAs we treat `*` and `#` as symbols. Recall that we compute the *reversed* DFAs. When we evaluate such an automaton on a path p, we break ties in the following order: we favor a constant label over `#` and `*`, and we favor `#` over `*`. Moreover, whenever we traverse a `#` transition, we append the corresponding label to the front of a buffer: if the automaton accepts, then the buffer holds at the end (an approximation of) $Match(p, c)$. For example, consider the automaton $A2$ in Fig.6 (in reverse), and the path `p = /apache:entry:/apache:date`. First, we traverse `apache:date`, which corresponds to `#` hence we store it in the buffer, next we traverse `apache:entry` (it takes precedence over `*`). The result for $Match(e2, p)$ is {`apache : entry`}. Finally, each value $Match(p, c)$ needs to be translated into a container number: this is done using an additional trie structure (details omitted).

5.3 Semantic Compressors

eat Semantic compressors in `XMill` conform to a C++ API called SCAPI (Semantic Compressor API), and users can add their own compressors/decompressors to be compiled into `XMill`. The user defines a compressor/decompressor by extending the three classes UserCompressorFactory, UserCompressor, and UserDecompressor. Several methods must be implemented for registering and creating compressors and decompressors (in UserCompressorFactory), for compressing given text values and storing them in containers (in UserCompressor), and for decompressing text values from a container and generating the XML text (UserDecompressor) . Note that semantic compressors and decompressors can maintain a state between the (de)compression of single text values.

All semantic compressors in `XMill` conform to a $C++$ API called SCAPI (Semantic Compressor API). Users can add their own compressors/decompressors to be compiled and linked into `XMill`, by conforming to this API. Fig. 9 shows the core of the API. For the type `bool` we use `unsigned char`. When defining a compressor/decompressor the user extends each of the three classes, and overloads their methods with her own implementation.

Class UserCompressorFactory is used for instantiating UserCompressor- and UserDecompressor-objects. Combi-

```
class UserCompressorFactory
{
   virtual char *GetName()=0;
   virtual UserCompressor   *InstantiateCompressor  (char *paramstr, int len)=0;
   virtual UserDecompressor *InstantiateDecompressor(char *paramstr, int len)=0;
};
class UserCompressor
{
   virtual unsigned GetContainerNum()=0;
   virtual bool     IsRejecting()=0;
   virtual bool     ParseString   (char *str, int len, void *state)=0;
   virtual void     CompressString(char *str, int len, Container *contarray, void *state)=0;
   virtual void     FinishCompress(Container *contarray,void *state)=0;
};
class UserDecompressor
{
   virtual void     DecompressItem(DecomprCont *contarray, XMLOutput *output, void *state)=0;
};
```

Figure 9: The SCAPI-interface (fragment) for implementing semantic compressors

nator compressors[15] like `seq(...)` or `or(...)` also require a parameter string. For example when seq(u8 "-" u8) occurs on the command line, then seq is called with the string u8 "-" u8. For atomic compressors, like u8, the parameter string is empty. Each compressor factory must also have a name (GetName()) that uniquely identifies the compressor.

The UserCompressor-object must determine how many subcontainers it needs (e.g. seq(u8 "-" u8) needs two) and whether it can reject a string (e.g. u8 could reject string, while e does not). Rejection in the context of composed compressors is more difficult, since we need to undo the compression actions of previous subcompressors. To avoid that we decided on a two-step strategy, where parsing is done in the first step (ParseString), and compression in the second (CompressString): rejection is decided during the first step, when no compression took place. Semantic compressors can maintain their own state, given by the parameter state. This is important for stateful compressors such as the run-length and delta encoders.

Finally, the UserDecompressor-instance provides a method to decompresses the given data from source containers and to write the decoded string to output stream output.

We omit several technical details about the SCAPI-interface, such as the initialization and memory management.

6 Experimental Evaluation

We evaluated `XMill` on several data sets. Our goal was to validate `XMill` for XML data archiving and data exchange. We also wanted to test `XMill`'s feasability as a compensatory tool for migrating other special-purpose data formats to XML.

Data sources We report the evaluation of `XMill` on six data sources, whose characteristics are shown in Fig. 6. The Weblog and the SwissProt data were described in Sec. 2; from the original SwissProt data we eliminated the DNA sequences. Treebank [25] is a large collection of parsed English sentences from the Wall Street Journal stored in a Lisp like notation, which we converted to XML (see Sec. 5). TPC-D(XML) is an XML representation of the TPC-D benchmark database, using two levels of nesting[16]. We deleted from the TPC-D data the `Comment` field, which takes about 30% of the space, and consists of randomly generated characters. DBLP is the popular database bibliography database[17], and is stored in a large collection of small XML files, which we concatenated

[15] Users can define their own combinator compressors.
[16] We tried other XML representations too, and observed no significant change in the experimental results.
[17] `http://www.informatik.uni-trier.de/ ley/db/index.html`

Data Source	Original Size	Size in XML	Regular?	Depth	Tags	DataGuide Size
Weblog Data	57.7MB	172MB	yes	1	10	11
SwissProt	98.5MB	158MB	yes	3	92	58
Treebank	39.6MB	53.8MB	no	35	251	339920
TPC-D	34.6MB	119MB	yes	2	43	60
DBLP	-	47.2MB	yes	3	10	145
Shakespeare	-	7.3MB	no	5	21	58

Figure 10: Data sources for performance evaluation

into one large file. Finally, Shakespeare is a corpus of marked-up Shakespeare plays, and it is stored directly in XML.

Figure 6 shows the size of the original data sources, the size of their XML representation, and four characteristic measures: our assesment of the data's regularity (yes/no), the maximal depth of the XML tree, the number of distinct tags, and the number of nodes in the DataGuide (another measure of (i)regularity).

Classes of experiments For each data source, we performed three classes of experiments. First, we compared the compression ratios of `gzip` and `XMill` under various settings. When available, we also applied `gzip` to the original format. We also tested the variation of the compression ratio as a function of the data size, and its sensitivity to the memory window. Second, we measured the compression and decompression times of `XMill` and `gzip`. Third, we measured the total effect of `XMill` in an XML data exchange application over the network.

Platform We ran the first two sets of experiments on a Windows NT machine with a 300MHz Pentium Processor and 128MByte main memory. The data exchange experiment was performed by sending data from AT&T Labs, running an SGI Challenge L (4 x 270MHz MIPS R12000, Irix 6.5.5m) to two places: the University of Pennsylvania, running a Sun Enterprise 3000 (4 x 250Mhz UltraSPARC) with 1024MB of memory, and a home PC (100MHz, Linux) with 32MB of memory connected to a cable modem. We transfered files with `rcp`[18], for which we measured a transfer rate of 8.08MBits/s (AT&T to Penn) and 1.25MBits/s (AT&T to home PC via cable modem).

Experimental Methodology The *compression ratio* is expressed as "bits per bytes". For example, 2 bits/bytes means that the compressed file size is 25% of the uncompressed file size (lower is better). The *running time* represents the elapsed time in seconds. A significant portion is spent reading and/or writing a file, and we observerd a high variance in the first 2-3 runs, while the operating system buffers are initialized, and after that a decrease and stabilization of the running time. Every data point is obtained by running the experiment eight times and taking the average of the last five runs. For the data exchange experiment, we measured the compression and decompression times separately (at AT&T) from the data transfer; each was executed eight times, as explained.

In comparing the running time of `XMill` with `gzip` we noticed significant differences in the efficiency of `gzip` (the stand-alone tool) and `zlib` (the library function, which is used in `XMill`). Interestingly, these differences depend on the platform: `zlib` is twice slower than `gzip` on NT, while being about 40% faster on an SGI. There is also a tiny difference (< 1%) in compression ratio, but the binary formats are compatible. For meaningful comparisons, we replaced `gzip` with `minigzip`, a stand-alone program included in the `zlib` library package. We compiled `minigzip` with exactly the same options as `XMill`. In all experiments below, "`gzip`" actually means `minigzip`.

6.1 Compression Ratio

Figure 11 shows the compression ratios for different data sources and compressors. For each data set, the four connected bars represent `gzip`, and `XMill` run with three settings (as in Sec. 2): no grouping (`XMill //`), grouping based on parent tag (`XMill //#`; this is the default setting), and user-defined grouping with semantic

[18]We also tried `ftp`, which was much slower, hence more favorable to `XMill`, and `lynx`, which had about the same transfer rate as `ftp`.

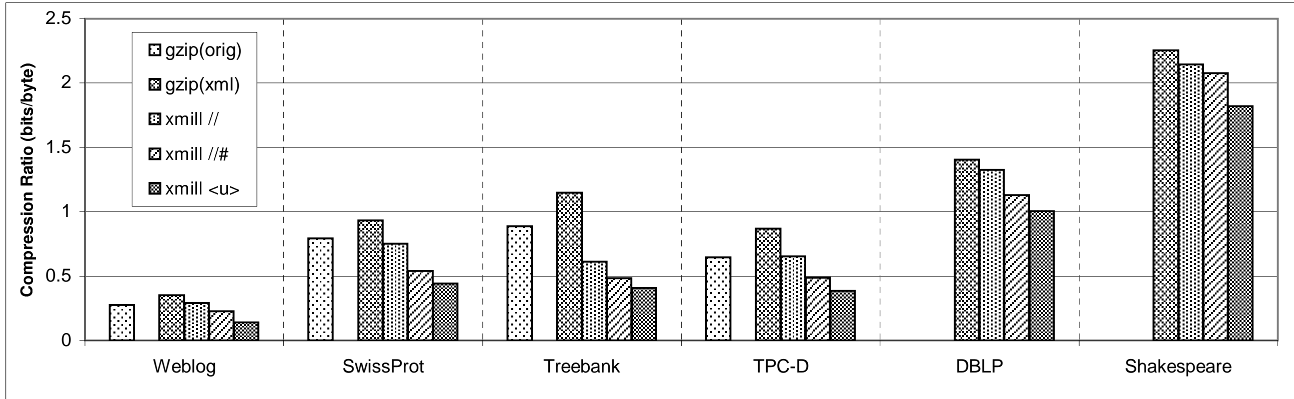
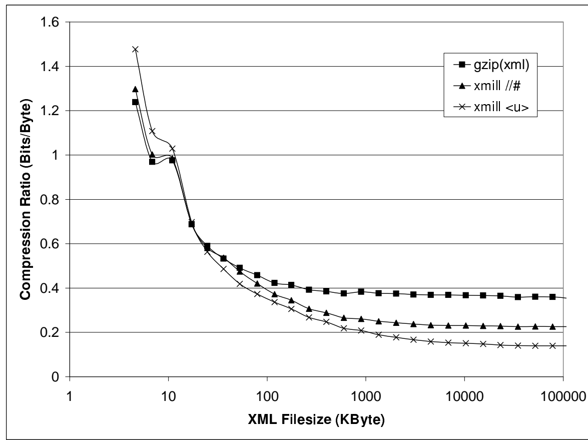
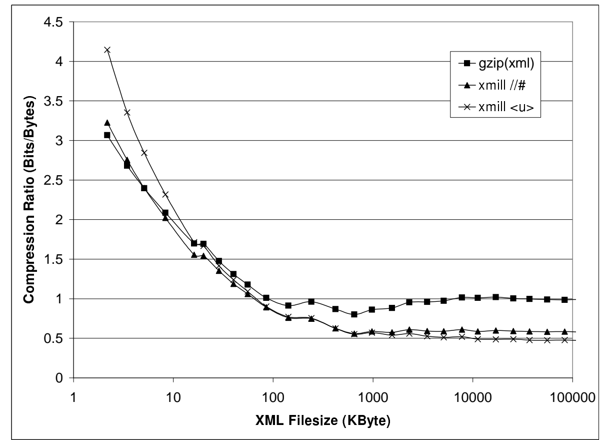


Figure 11: Compression Results



(a) Weblog



(b) SwissProt

Figure 12: Compression Ratio under Different Sizes of Weblog and SwissProt

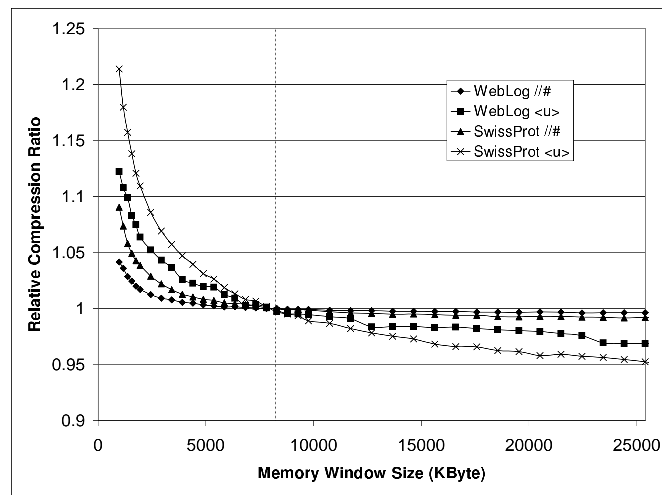


Figure 13: Relative Compression under Different Memory Windows

compression (abbreviated XMill <u>). Here and in subsequent experiments, in XMill <u> we used the best combination container expressions we could find for each particular data set. As expected, better settings for XMill always produced better compression. For the first four data sets (which had more data and less text), XMill's compressed under the default setting to 45%-60% the size of gzip: using semantic compressors, XMill reduced the size to 35%-47% of gzip's. For the more text-like data sets, XMill still performed better than gzip, but less spectacularly. For the first four data sets, the bar on the left represents the size of the gzipped original file (i.e. the height of the bar is size(gzip(orig))/(8*size(XML))). With the default setting, XMill already compressed better the XML file than gzip compressed the original file. With user compressors, XMill compressed to a fraction of about 45% to 60% of the size of the gzip'ed original file.

Fig. 12 shows the variation of the compression ratio as a function of the XML data size for Weblog and SwissProt. Here we ran the compressors on several subsets of the two data sets. On small files XMill performs worse than gzip because it splits the data into too many small containers, on which compression is ineffective. The crossing point in both cases was at about 20KB. Note that, although the main memory window is fixed to 8MB, much more of the XML data gets compressed in one block, because most of the space in XML is taken by the tags which are dictionary encoded. The 172MB Weblog data is compressed in 4 blocks by XMill <u>, and in 9 blocks by XMill //#. This explains why the compression rate continues to improve beyond data of 8MB.

Fig. 13 shows the sensitivity of XMill's compression ratio on the memory window size. The compression ratio is normalized with respect to a window size of 8MB, XMill's default. The results show that a smaller memory window size substantially degrades the compression rate, again because a small window implies small containers, on which the compression rate is poor. Beyond 8MB both data sets were compressed in only a few blocks, and the compression rate did not improve too much.

6.2 Compression/Decompression Time

We measured the compression and decompression time for three data sources: Weblog, SwissProt, and Treebank and three compression strategies: gzip, XMill //#, and XMill <u>[19] Figure 14(a) shows the compression time for each data source and compressor. For XMill, the time is split into two parts: (1) parsing and applying semantic compressors, and (2) applying gzip. Overall XMill is generally as fast as gzip. XMill basically saves time by applying gzip to a smaller data size and spends the time on regrouping to improve the the compression rate. For the same reason, XMill <u> is faster than XMill //#, because the semantic compressors pre-compress the data, hence gzip spends less time.

Figure 14(b) shows the decompression time for each of the data sources and compressors, broken down according to the decompression step. There are four such steps: (1) gunzip the containers, (2) interpret the XML structure and merge the data values, applying the appropriate semantic decompressors: this results in a stream of SAX events, (3) generate the XML string (start-tags, end-tags, data values, etc), (4) output the file. For gunzip we only have steps (1) and (4). Note that the time fragmentation into parts (1)-(4) is not completely accurate, because of caching interferences.

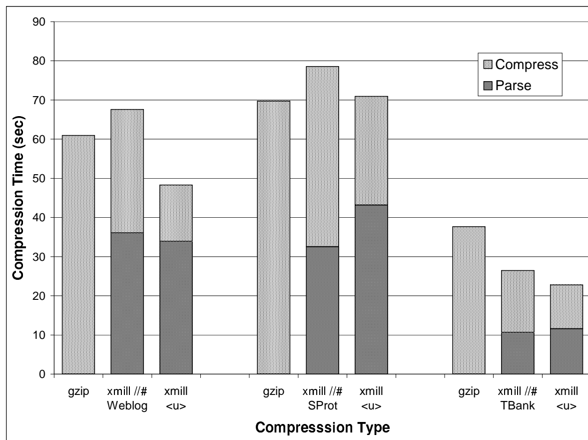
For a complete decompression (written to a file) XDemill's speed is comparable to gunzip. Some applications however would decompress on-the-fly, and do not need the output step. If we remove the output step (4) (which accounts for most of the time), then XDemill is about twice slower than gunzip. We pay here the price of having to merge data from different containers into one single string. However, an application could do even better by consuming SAX events directly, rather than having to re-parse the XML string: such applications only need XMill to perform steps (1) and (2) (gunzip's output always needs to be parsed).

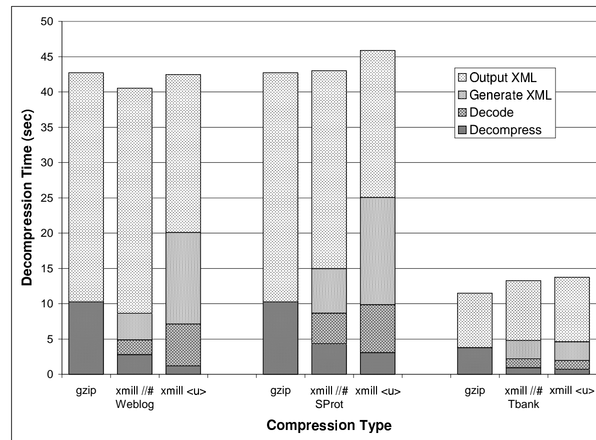
Note that the compression and decompression time is typically linear in the size of the data. Hence, we do not show a diagram similar to Fig. 12.

6.3 Data Exchange

Fig. 15 shows the results of exchanging Weblog data from AT&T to the University of Pennsylvania (a) and to a home computer via a cable modem (b). Recall that the transfer rate via the cable modem was much lower (1.25MB/s v.s. 8.08MB/s), which favors XMill because of the better compression rate. The bars are split into compression time (lower bar), and transmission+decompression (upper bar). The end-to-end transfer time (from XML file to XML file) is dominated by the compression time: here there are no significat differences between

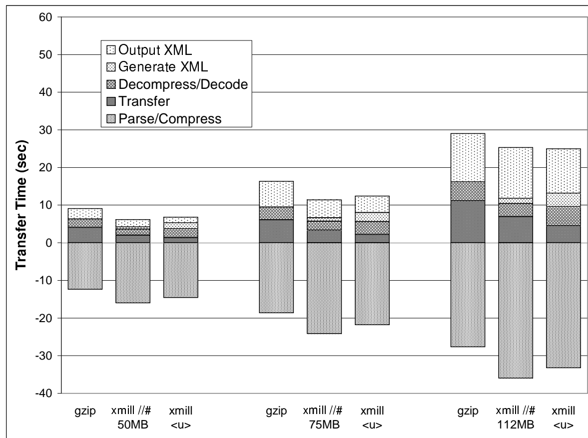
[19]The compression/decompression time of XMill // is comparable to XMill //#.

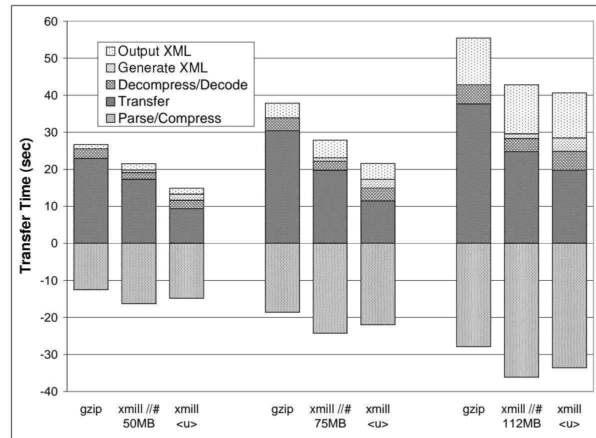




(a) (b)

Figure 14: Compression (a) and Decompression (b) Time





(a) (b)

Figure 15: Network Transfer Time from AT&T Labs to Penn (a) and to a home PC (b)

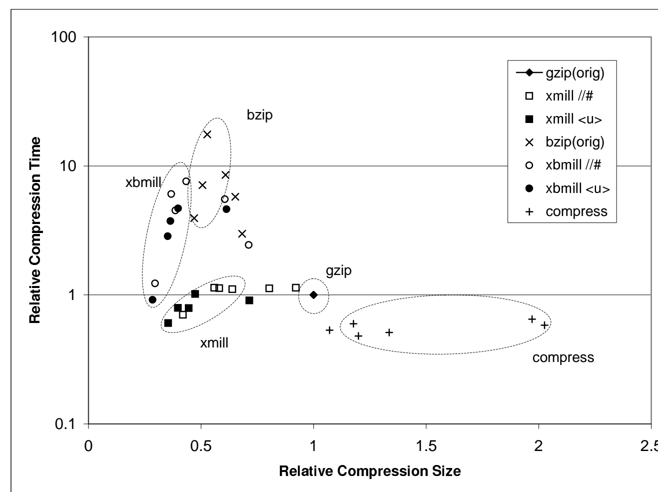


Figure 16: Comparing Compression Rate and Compression Time

XMill and gzip (with some slight advantage for XMill on the slow network). If the file is already compressed, then XMill shows detectable improvements over gzip; again, better so in the case of a slow network. Furthermore, if the decompressed data is used directly in an application (via a SAX interface), then only the transfer and decompress+decode time matters, and XMill is a clear win over gzip: it takes only about 60% of the time. We also ran the exchange experiments for the SwissProt data and obtained similar results that are omitted.

7 Discussion and Future Work

Benefits of XMill The experiments show that XMill clearly achieves better compression rates than gzip (around a factor of 2, for data-like XML, less for text-like XML), without sacrificing speed; below we compare XMill to other compressors and conclude the same. This makes XMill a clear winner for data archiving. For data exchange however, the picture is more complex. While XMill never looses to gzip, whether its improvements are significant or barely detectable depends on two factors: the type of exchange application, and the relative processor v.s. network speed. For a slow network, XMill's improvements are always detectable, because of its better compression rate. For a fast network, one has to look at all three exchange steps: compression, network transfer, and decompression. Compression is consistently the most expensive, and is about the same in gzip and XMill. Hence, the relative advantage depends on the type of application. For an end-to-end file transfer, there is no clear winner. In XML publishing the file is compressed only once, and only network transfer and decompression matters: XMill is consistently, but only modestly faster than gzip. If, moreover, the data is imported directly into applications, then the decompression does not need to produce an output XML file, but only to generate the SAX events: here XMill can become significantly better than gzip.

Of course, in applications where the cost of the network bandwith is much higher than processor cost, XMill is again a clear win, regardless of the other parameters.

For further improvements in data exchange, future work has to focus on the compression/decompression time, not the compression rate. We invested considerable effort to make the compresssion fast, and believe that no quick fixes can bring dramatic improvements (say, a factor of 2). One possibility is to replace zlib with a faster, but less performant compressor, like compress. That is, trade off space for time. We realized late into the project the value of such a tradeoff in data exchange, and did not have time to pursue this direction. However, zlib accounts for only about 50% of the compression time (Fig. 14), so such a change needs to be complemented with more improvements in the path processor. For decompression, the bottleneck is XDemill's need to merge data from several containers while interpreting the structure. One possible improvement is to have a more concise structure with "supertokens": this would reduce the interpretation time.

Time/Space Tradeoff Many different general-purpose compressors exists, offering a variety of time/space tradeoffs. We tried a few of them on our six data sets (Fig. 6): gzip, compress, and bzip, where compress is faster than gzip but achieves worse compression rates, while bzip achieves better compression rates but is excessively slow. The results are shown in Fig. 16, where all compression rates and compression times are normalized wrt. that of gzip. The blobs highlight the the "data-like" XML data sets (Weblog, SwissProt, Treebank, and TPC-D). The diagram shows clearly that XMill offers the best overall time/space tradeoff for XML data. Given bzip's impressive performance, we tried to replace gzip with bzip in our compressor XMill. As expected, the resulting compressor (called xbmill) compresses better than XMill. Interestingly, the compression times did not increase as badly as between gzip and bzip: this is because bzip's running time is more than linear (it sorts the input data), and therefore is much faster when applied to small containers (in xbmill).

Schema Extraction At this point in our project all container expressions in XMill have to be specified manually. They were designed keeping the XML-Schema in mind [33, 5], and it is relatively straightforward to generate them from a given XML-Schema. However, it would be much more useful to extract them automatically from a given XML data set. Unlike previous work on schema extraction for semistructured data [26] which focuses on the data's graph structure, in XMill the critical part is choosing the right semantic compressor for each container. An automatic tool would have to recognize an integer field, a date field, or a structured field: data mining techniques could be deployed for this purpose.

8 Related Work

General Compression General compression methods are described in textbooks [4, 30]. A more recent method is block-sorting compression described in [8], which sort the characters in a file first, before applying LZ77. This tends to group repeated sequences together, offering more oportunities for a dictionary-based compressor. The algorithm uses a very clever technique to recover the original order at decompression time. The Unix tool `bzip` (`http://www.bzip2.org`) is based on this method.

Database Compression In databases, compression has been advocated as a method for cost reduction: to save storage space and improve processing time, based on the observation that much of the query processing time is due to I/O. Hence, the compression techniques are designed to allow the query processor to decompress a small unit of data at a time: one column value in the table, or one row. A survey of compression techniques in databases can be found in [29]. More recent work has proposed specialized compression techniques, which we briefly mention here.

Iyer and Wilhite [23] propose a new variant of Ziv-Lempel's algorithm to compress one row at a time. The variant is non-adaptive, in order to make decompression of one row possible. A fixed size dictionary is constructed once for the entire database (using sampling), then each row is compressed individually, using the dictionary. The authors report reductions to about 30%-60% of the original size on several real data sets. Importantly, the response time for a table scan reduces to about 50%.

Goldstein et al. [19] propose a simple compression method to be used in conjunction with a database management system. Their compression is essentially a dictionary encoding, which uses a different dictionary for each page. A column may have a large number of values, and still be compressed well if its range in each page is small. A greedy grouping method is used to cluster tuples into pages.

Ng and Ravishankar [27] describe another specialized compression method called *tuple differential coding* (TDC). First, each column in the table is dictionary encoded, thus mapping each tuple into a point in a hyperrectangle with k dimensions, where k is the number of columns. Further, the points in this hyperrectangle are sorted lexicographically, and enumerated: each tuple now becomes a number. Tuples are sorted in increasing order of their number, then differential-encoded.

Two features distinguish `XMill` from this previous work: `XMill` is not designed to be used in a query processor, and we do not propose a new compression algorithm, but rather offer a framework in which existing algorithm can be leveraged to compress XML data.

Semistructured Data Query Languages Various query languages for semistructured data have been considered [1, 7, 15], and all feature some forms of regular path expressions. XPath [11] is part of the XSL W3C recommendation. Since our container expressions are used in the command line, we decided on a syntax resembling XPath because of its conciseness.

XML Storage Several ways have been proposed to store XML data in relations [14, 16, 31]. In all these approaches a critical issue is the ability to reconstruct the original XML data in a declarative way, so that queries over the XML view can be translated into SQL queries on the relational store. In some sense, `XMill` also separates the XML data values into columns (a container = a column). The difference is that we can use an imperative program (`XDemill`) to reconstruct the XML data: this enables us to do a much more aggressive mapping into containers.

Other XML Compressors Several companies or organizations seem to be working on XML compression: this is suggested to us by the fact that the domain names `www.xmlzip.com`, `www.xzip.com`, `www.xmlcompress.com`, and `www.xcompress.com` have all been registered. At the time of writing, a single product has been announced, by XML Solutions, called `xmlzip` (`www.xmlzip.com`). Implemented in Java, `xmlzip` cuts the XML tree at a certain depth and compresses the upper part separately from the lower part, both using `gzip`. Tested on our data sets (Fig 6), it ran out of memory on all sets except Shakespeare. There, it achieves a compression ratio between that of `gzip`'s and `XMill`, but at much lower speed. Although `xmlzip` is not a serious challenge to `XMill`, it impacted our project seriously because our initial choice for a name was `xmlzip`, and we had to abandon it.

A compressor for tabular data: `pzip` An interesting tool which influenced us during this project is `pzip` [3]. It compresses files with fixed-length records 2 to 10 times better than `gzip` at 2 to 3 times the speed (both during compression and decompression), using the `zlib` library function. These are quite impressive numbers ! But one should add that many of the improvements come from eliminating the redundancies introduced from making the records fixed length: many columns are mostly blanks (or zero), and `pzip` uses run-length encoding on these. By contrast [23] argues that since modern database systems support variable-length records, run-length encoding is less beneficial. What makes `pzip` especially interesting is its schema extraction tool: using a window from the data, it discovers the columns with low entropy (on which it applies run-length encoding), while for the others it finds the best way to group them before submitting to `zlib`.

9 Conclusions

We have described a compressor for XML data called `XMill`, which is an extensible tool for applyin existing compressors to XML data. Its main engine is `zlib`, the library function variant for `gzip`. One of our targeted applications is XML data archiving, where compression rate counts alone. Here `XMill` achieves about twice the compression rate of `gzip`, at roughly the same speed, and it generally ranked best among a few other compressors we compared it against: `compress`, `bzip`, `xmlzip`. A second application we target is data exchange, where both compression ratio and compression/decompression time count. While `XMill` never looses to `gzip`, the size of its improvements depends on a variety of factors (type of application and relative processor/network speed), and range from none to almost a factor of 2.

Acknowledgments We would like to thank Ken Church, Mary Fernandez, Glenn Fowler, and Val Tannen for their helpful comments. Special thanks to Peter Buneman who suggested to us the name `XMill`.

References

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1996.

[2] L. Allison, T. Edgoose, and T. I. Dix. Compression of strings with approximate repeats. In *Proceedings of International Conference on Intelligent Systems for Molecular Biology*, pages 8–16, Montreal, Canada, June 1998.

[3] D. Belanger and K. Church. Data flows with examples from telecommunications. In *Proceedings of 1999 Workshop on Databases in Telecommunication*, Edinburgh, UK, September 1999.

[4] T.C. Bell, J.G. Cleary, and I.H. Witten. *Text Compression.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[5] P.V. Biron and A. Malhotra. XML schema part 2: Datatypes. *W3C Working Draft*, September 1999. Available as `http://www.w3.org/TR/xmlschema-2`.

[6] P. Buneman, S. Davidson, Mary Fernandez, and D. Suciu. Adding structure to unstructured data. Technical Report MS-CIS-96-21, University of Pennsylvania, 1996.

[7] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 505–516, Montreal, Canada, June 1996.

[8] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Equipment Corporation, May 1994.

[9] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogenous information sources. In *Proceedings of the Information Processing Society of Japan Conference*, Tokyo, Japan, October 1994.

[10] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In Richard Snodgrass and Marianne Winslett, editors, *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota, May 1994.

[11] J. Clark and S. DeRose. XML path language (XPath), version 1.0. *W3C Working Draft*, August 1999. Available as `http://www.w3.org/TR/xpath`.

[12] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversation ! In *ACM-SIGMOD International Conference*, pages 177–188, Seattle, Washington, June 1998.

[13] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for xml. In *Proceedings of the Eights International World Wide Web Conference (WWW8)*, Toronto, 1999.

[14] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *ACM SIGMOD International Conference on Management of Data*, Philadelphia, May 1999.

[15] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel. In *Proceedings of ACM-SIGMOD International Conference*, pages 414–425, Seattle, Washington, June 1998.

[16] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *Data Engineering Bulletin*, 22(3), 1999.

[17] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the lore data model and query language. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, pages 25–30, Philadelphia, PA, June 1999.

[18] R. Goldman and J. Widom. DataGuides: enabling query formulation and optimization in semistructured databases. In *Proceedings of the International Conference on Very Large Data Bases*, pages 436–445, Athens, Greece, August 1997.

[19] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *Proc. IEEE Conf on Data Engineering*, 1998.

[20] S. Grumbach and F. Tahi. A new challenge for compression algorithms: genetic sequences. *Information Processing and Management*, 30(6):875–886, 1994.

[21] D. G. Higgins, R. Fuchs, P. J. Stoehr, and G. N. Cameron. The EMBL data library. *Nucleic Acids Research*, 20:2071–2074, 1992.

[22] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 1979.

[23] B.R. Iyer and D. Wilhite. Data compression support in databases. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases*, pages 695–704, Santiago de Chile, Chile, September 1994.

[24] T. Lahiri, S. Abiteboul, and J. Widom. Ozone - integrating semistructured and structured data. In *8th International Workshop on Database Programming Languages (DBPL)*, Kinloch Rannoch, Scotland, September 1999.

[25] M.P. Marcus, B. Santorini, and M. Marcinkiewicz. Building a large annotated corpus of english: the penn treebank. *Computational Linguistics*, 19, 1993.

[26] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring structure in semistructured data. In *Proceedings of the Workshop on Management of Semi-structured Data*, 1997. Available from `http://www.research.att.com/~suciu/workshop-papers.html`.

[27] W.K. Ng and C.V. Ravishankar. Block-oriented compression techniques for large statistical databases. *TKDE*, 9(2):314–328, 1997.

[28] D. Perrin. Finite automata. In *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 1, pages 1–57. Elsevier, Amsterdam, 1990.

[29] M. A. Roth and S. Van Horn. Database compression. *ACM SIGMOD Record*, 22(3):31–39, Sept. 1993.

[30] D. Salomon. *Data Compression. The Complete Reference.* Springer, New York, 1997.

[31] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 302–314, Edinburgh, UK, September 1999.

[32] C.E. Shannon. A mathematica theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948. Also available in *Claude Elwood Shannon, Collected Papers*, N.J.A.Sloane and A.D.Wyner eds, IEEE Press, 1993.

[33] H.S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML schema part 1: Structures. *W3C Working Draft*, September 1999. Available as `http://www.w3.org/TR/xmlschema-1`.

[34] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.